

Destruction derby

If your hard disk is rapidly filling up with files and you're losing control, don't panic. **Mike James** shows you how to write a program that will automatically delete files after a set time



FAQ

Q What is a memory leak?

A A memory leak occurs when an application accidentally holds on to memory that it no longer needs. For example, when you create a string or any variable or object, memory is allocated to store it. The amount of memory can grow and shrink as the amount of data stored changes. If the unused memory isn't reclaimed, this is a memory leak.

Usually the memory that a variable or object uses is reclaimed when it is no longer needed, or when it reaches the end of its life and is destroyed. This is called garbage collection and is handled by the system.

If the system leaves dead copies of objects in memory, your application uses increasingly large amounts of memory. Fortunately, memory is returned for use when your application terminates.

Mike James

IT author, lecturer
and programmer

mike@computershopper.co.uk

Hard disk capacities might be getting bigger, but in many ways this just increases the amount of useless data we store. The result is that data is harder to find, and there's more junk to sort out when you run out of disk space. The answer is to delete unwanted and duplicate information.

Of course, this is easier said than done, and it's hard for a computer to decide which files are unwanted clutter and which are incredibly valuable.

With help, though, a computer can make the job easier. For example, you might want to keep personal information for a set amount of time before getting rid of it. Also, if you're working on a project, you might want to set its temporary files to be deleted a month after you've completed it.

This month we'll show you how to write a program that lets you select files and choose their deletion time. The techniques we teach you can be expanded to different criteria, so you could write your own program that deletes files based on different factors.

SEND TO

The first problem is to find a way of enabling the user to specify the 'time to live' for a set of files. There are many ways of doing this, and the ideal solution would allow the user to set a timed delete easily and naturally when they first create the document. Unfortunately, this is a tall order because it would mean writing an add-on for every application in use. A more general approach isn't as convenient to use but is easier to implement and provides a starting point for more specialised implementations. The easiest way to provide a custom file command is to use the Send To right-click menu option.

Compared with other custom menu options in Windows, the Send To menu is very easy to modify. For each user there is a special directory called:

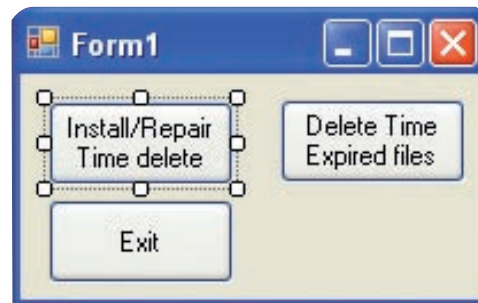
Documents and Settings\user name\Send To

Any shortcuts that you store in this directory automatically appear in the Send To sub-menu as options. It really is that simple. When the user selects a set of files (all in the same directory), right-clicks and selects the shortcut from the Send To sub-menu, they start the application that the shortcut targets. The full path names of all the selected files are provided to the application through a set of command-line arguments.

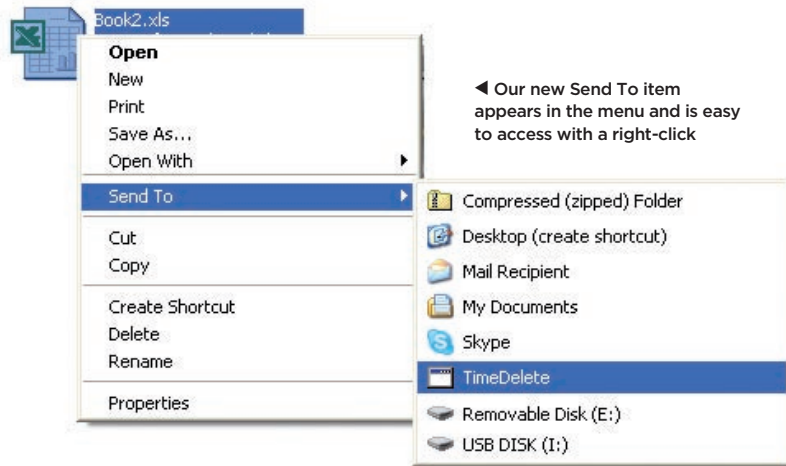
Installing a Send To application is simply a matter of creating a shortcut, and this is so easy it hardly seems worth inventing a separate installation module. So with this in mind, it's worth creating a single .EXE file that does everything, including installation. This way, distributing the final application is just a matter of downloading and running a single .EXE file.

GETTING STARTED

We'll use C# for this project. You should be able to use any Express or full edition of Visual Studio, as no recent .NET features are required. For the same reason conversion to any other .NET language is very easy.



▲ To start the project, switch to the form editor and add three buttons with the captions shown here



Start a new project called TimeDelete. Switch to the form editor and add three buttons with the captions 'Install/Repair Time delete', 'Delete Time Expired Files' and 'Exit'.

The first code to create is the installation procedure, which will make a shortcut to the application in the Send To folder. First, we need the location of the Send To directory for the current user and the current directory in which the executable is stored. Both are easy to get, with the help of the Environment class:

```
private void button1_Click(object sender, EventArgs e)
{
    string SendTo =
        Environment.GetFolderPath(
            Environment.SpecialFolder.
                SendTo);
    string ProgramPath =
        Environment.CurrentDirectory;
}
```

Now we are ready to create the shortcut, but there is a problem. Even though there are lots of helpful classes in the .NET Framework, there doesn't seem to be one that lets you work with shortcuts. There are a number of solutions to this problem, including using `plnvoke` to call low-level API functions, but arguably the simplest is to use an ActiveX class library that is generally available as part of the Scripting support (for both JScript and VBScript). To use it, first load a reference to the COM object, Windows Script Host Object, and add:

```
using IWshRuntimeLibrary;
```

Now we can create a shortcut in exactly the same way we would using VBScript or JScript. First create a `WshShell` object:

```
WshShellClass WshS = new
WshShellClass();
IWshShortcut Shortcut =
    (IWshShortcut)WshS.
        CreateShortcut(
            SendTo + @"\TimeDelete.lnk");
Shortcut.TargetPath =
    ProgramPath +
        @"\TimeDelete.exe";
Shortcut.WindowStyle = 7;
Shortcut.Save();
}
```

At this point we've added a shortcut that adds the TimeDelete option to the Send To menu.

Before moving on to the next stage, it's worth creating the code for the Exit button:

```
private void button2_Click(object sender, EventArgs e)
{
    Application.Exit();
}
```

THE USER INTERFACE AND DESIGN

Next we have to implement some code that allows the user to specify the lifetime of a file. There are many possible ways of doing this. We could add a task to the scheduler that deleted the file at the specified date. We could store a 'delete time' property with every file and then periodically scan for files that have expired.

Each approach has its advantages and is implemented on a different scale. The approach we've selected here won't suit everyone but it scales well and won't accidentally clog a machine. Consider what happens, for example, when a computer that has been switched off for three months is turned on for the first time and starts to deal with perhaps thousands of files that need deleting. A safer approach is to store the details of the files in a text file and allow the user to choose when to process expired files.

The first thing to do is to add a label, a list box, a selection box and two buttons to the form with captions 'OK' and 'Cancel'. Add some time periods, such as '1 Day' and '1 Week', to the selection box. Arrange the new controls beneath the existing three buttons.

When you are happy with the arrangement, select all the new items and drag them up the form to cover the existing three buttons. This may seem like madness, but we are going to add some code that hides the buttons that aren't



going to be used in a particular mode of operation. Resize the form to fit the newly arranged controls.

We need to do some initialisation in the form's constructor for the new controls, and we also need to store the arguments passed to the application:

```
private string[] m_args;
public Form1()
{
    InitializeComponent();
    m_args = Environment.
        GetCommandLineArgs();
    listBox1.SelectedIndex = 2;
    checkBox1.Checked = true;
}
```

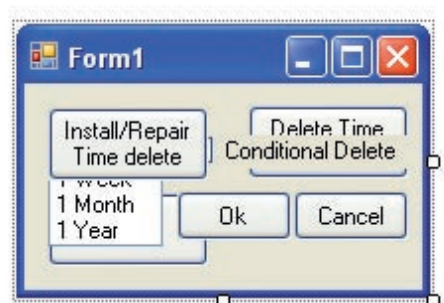
The choice of which part of the user interface to show can be made according to the number of arguments passed to the application. If the application is passed more than one, the user is adding files to the list to be deleted and we need to see only the new controls. We can hide the controls we don't need to see as part of the Shown event:

```
private void Form1_Shown(object sender, EventArgs e)
{
    if (m_args.Length == 1)
    {
        button4.Hide();
        button5.Hide();
        checkBox1.Hide();
        listBox1.Hide();
        label11.Hide();
    }
    else
    {
        button1.Hide();
        button2.Hide();
        button3.Hide();
    }
}
```

With this addition the user interface presents itself correctly to the user.

THE DELETE LIST

Now we need to record the files that the user wants to delete. When the user selects a group of files and sends them to the TimeDelete application, it starts running with an array of arguments that specify each file. The user is presented with the form and can select the time to delete and whether or not an 'are you sure?' prompt is shown before the file is deleted. They



First lay out the new controls and then place them on top of the existing buttons



▲ The Send To menu is very easy to modify: simply place a shortcut in the Send To directory

then click the OK button and the application has to create a record of those files.

An early design decision to store the text file that lists the files to be deleted in the same directory as the application caused a problem at first. When the application is run from the Send To menu its current directory is set to the one above the directory in which the files are stored and not the directory from which it was started. The solution is to read the target directory information from the link stored in the Send To directory. First we get the Send To directory:

```
private void button4_Click(object sender, EventArgs e)
{
    string SendTo =
        Environment.GetFolderPath(
            Environment.SpecialFolder.
            SendTo);
}
```

Next we get the shortcut:

```
WshShellClass WshS = new
WshShellClass();
IWshShortcut Shortcut =
(IWshShortcut)
WshS.CreateShortcut(SendTo +
@"\TimeDelete.lnk");
```

People often don't appreciate that if you create a Shortcut object in this way it is loaded with all the data in any existing shortcut. This means we can easily get the target directory:

```
string ProgramPath = System.IO.Path.
GetDirectoryName(Shortcut.
TargetPath);
```

Now that we have the location we can begin to process the data. First we need to work out the date on which the files need to be deleted. We need an array for the number of days corresponding to each possible item in the listbox:

```
int[] intervals = { 1, 7, 30, 365 };
```

There is scope for customising this, but adding one day, week, month and year seems to be a reasonable initial choice. Next we need to build up a string that contains a filename, a date to delete and a prompt for each file to be added to the data file. The most efficient way of doing this is to use a StringBuilder object:

```
StringBuilder buffer =
new StringBuilder(1024);
```

We also need to process each argument in turn:

```
for (int i = 1; i < m_args.Length;
i++)
{
```

It is easy to construct a TimeSpan object set to the number of days to deletion:

```
TimeSpan interval = new TimeSpan(
intervals[listBox1.SelectedIndex],
0, 0, 0, 0);
```

To get a delete date we need to add this TimeSpan to the current date (DateTime.Now) and build up the line specifying what to delete and when in the StringBuilder:

```
buffer.Append(m_args[i] + ", " +
DateTime.Now.Add(interval).
.ToShortDateString() + ", " +
checkBox1.Checked.ToString() +
Environment.NewLine);
```

This looks like a horribly long and complicated line of code, but it's very simple if you take it a chunk at a time. The first item is `m_args[i]`, which is just the complete path to the file to be deleted. The second item is the date and time with the TimeSpan object added to it and converted to a short date string. Finally we have the True or False corresponding to the state of the checkbox. Each item is separated by a comma, and the line ends with a `NewLine`. When the for loop ends, the StringBuilder object has such a line for each file that the user selected. Now that everything is ready, we can write the new data to the data file, which is very easy:

```
}
System.IO.File.AppendAllText(
ProgramPath + @"\DeleteData.
TDD",
buffer.ToString());
Application.Exit();
}
```

The static File object has a number of methods that are very easy to use and make it quick to build an application that uses files to store and retrieve data. The `AppendAllText` method either creates a new file or, if it already exists, opens the existing file and stores all the data in the specified string at the end of the file. You don't even have to bother closing it.

DOING THE DELETE

The user has to run the TimeDelete application directly to scan the list of files and delete any that have expired. The first thing we need is the path to the data file. This is a matter of reading the shortcut again to obtain the target path:

```
private void button3_Click(object sender, EventArgs e)
{
    string SendTo =
        Environment.GetFolderPath(
            Environment.SpecialFolder.
            SendTo);
    WshShellClass WshS = new
    WshShellClass();
    IWshShortcut Shortcut =
    (IWshShortcut)WshS.
    CreateShortcut(SendTo + @"\
    TimeDelete.lnk");
    string ProgramPath = System.
```

```
IO.Path.GetDirectoryName(
Shortcut.TargetPath);
```

Now that we have the path we can open the data file. This time we have no choice but to use a more standard approach to file handling and use a Streamreader, but the File object still provides an easy way to get one:

```
System.IO.StreamReader FileList=
System.IO.File.OpenText
(ProgramPath +
@"\DeleteData.TDD");
```

We also need to know the current date to compare it with the delete dates in the file:

```
DateTime now = DateTime.Now;
```

The simplest way of scanning the file is to use a do loop with a test at the end for the end of file condition:

```
do
{
```

Reading in a line and parsing it into its separate items couldn't be easier:

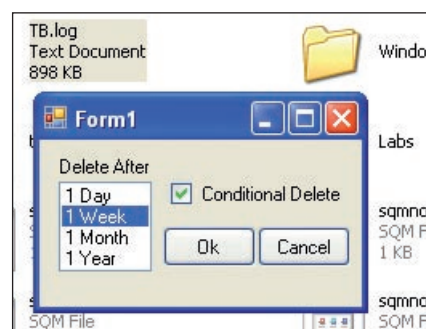
```
String[] Line = FileList.
ReadLine().Split(',');
```

The Split method returns a string array with each item in a separate element. Element [1] specifies the delete date, and it's easy to convert this to a date and test it against the current date:

```
DateTime DeleteTime = DateTime.
Parse(Line[1]);
if (now.CompareTo(DeleteTime) > 0)
{
```

If the file's lifetime is up we need to process it. The second element determines whether a prompt is to be issued or if the file should simply be deleted:

```
if (Line[2]=="True")
{
    DialogResult action = MessageBox.
    Show(
        "Delete " + Line[0] + "?",
        "TimeDelete",
        MessageBoxButtons.YesNo);
    if (action == DialogResult.Yes)
    {
        System.IO.File.
        Delete(Line[0]);
    }
}
```



▲ The listbox enables the user to specify how long the program should wait before deleting the files

PEACHPIT PRESS

Building a Web Site with Ajax

★★★★★

£7

From www.amazon.co.uk

Ajax stands for Asynchronous JavaScript and XML, and it's a hot topic at the moment. The title of this book is a little misleading, as it concentrates heavily on using PHP and MySQL. If you don't want to use these technologies, this book won't be much use to you.

If you are using them or are thinking about using them, this slow-paced and very clear step-by-step guide will help you to understand how everything works. It even deals with the often-ignored server side of development. It's a slim book, though, and with the use of big print and lots of illustrations, it doesn't get very far.

However, if you're keen to find out what all the fuss is about with Ajax, this is a great place to start. It takes you through a simple example that involves getting some data from the server and displaying it in a form using Ajax techniques to create a responsive program. The example may be narrow but it's the core of all other Ajax applications and it's just a matter of customising and extending it. If you are a beginner, this book is a great introduction to the topic.

SUMMARY

- ✓ Great for beginners
- ✗ Narrow focus

ISBN: 0321524411
PAGES: 176

APRESS

Beginning Game Development with Python and Pygame: From Novice to Professional

★★★★★

£15

From www.amazon.co.uk

Will McGugan's book is a little off the beaten track, as Python isn't a mainstream language and most game developers don't use it. This is a pity, as Python is a great language, and one that inspires a lot of enthusiasm among its fans.

If you know Python, you may worry that it's a little slow for games but Pygame is a library built on top of a C game creation facility that's fast enough for most things.

The book is very selective and concentrates on 3D games at the expense of 2D facilities. A possible weakness is that it lacks any convincing examples of a complete game.

If you're happy with small examples that demonstrate, among other topics, animation and collision detection, this is a fairly good book for Pygame. Some complete examples would make it an essential read, though.

SUMMARY

- ✓ Valuable Pygame reference
- ✗ No complete examples

ISBN 1590598725
PAGES 316

```
}
else
{
    System.IO.File.Delete(Line[0]);
}
}
```

Here we are using the File object to do the delete. If the file's time isn't up, we simply copy its data line to a new file called DeleteDate.T\$\$:

```
else
{
    System.IO.File.AppendAllText
        (ProgramPath +
        @"\\DeleteData.T$$",
        String.Join(",", Line) +
        Environment.NewLine);
}
} while (!FileList.EndOfStream);
FileList.Close();
```

We have to close the stream explicitly in this case. We have also used the Join method to rebuild the line from the separate elements in the string array. When the do loop comes to an end, all the lines of the file have been processed – the files have either been deleted or not according to what the user selected, and all the data lines corresponding to files that are still current have been written to the new file.

The final task is to juggle the filenames around so that the new file replaces the original. There are lots of ways of doing this, but bear in mind that you don't want to find yourself in a situation where there is no backup file on the disc, so one way of doing it is:

```
System.IO.File.Delete(ProgramPath +
    @"\\DeleteData.T$$");
System.IO.File.Move(ProgramPath +
```

```
@\\DeleteData.TDD",
    ProgramPath + @"\\
    DeleteData.T$$");
System.IO.File.Move(ProgramPath +
    @"\\DeleteData.T$$",
    ProgramPath +
    @"\\DeleteData.TDD");
```

In other words, first delete any existing file ending .T\$\$, then rename the .TDD file to .T\$\$ to create a new backup and, finally, rename the temporary .T\$\$ file to .TDD. In this way the user has a backup of the list in the .T\$\$ file, and during the swap over there is always one copy of the data left on the disk. You could try to work out the worst possible state the system could be left in if the process were to crash before completion, but to do the job properly takes a lot of effort.

THE FINER POINTS

You now have a complete program, but there are lots of things that can go wrong. In particular if the program attempts to delete a file that doesn't exist, it crashes. The solution here is to use exception handling. Change each of the delete commands:

```
System.IO.File.Delete(Line[0]);
```

to read:

```
try
{
    System.IO.File.Delete(Line[0]);
}
catch { }
```

The catch part of the expression usually contains instructions for dealing with the problem, but in this case we need only move on after failing to delete a file, because it has already been deleted.

You might like to add a message box that tells the user what has happened.

Another problem is in the block of code that renames the final files. The first time the program runs there isn't a .T\$\$ file, so the delete has to be surrounded in a try-catch:

```
try
{
    System.IO.File.Delete
        (ProgramPath +
        @"\\DeleteData.T$$");
}
catch { }
```

A much subtler problem is hidden in the line:

```
System.IO.File.Move(ProgramPath +
    @"\\DeleteData.T$$", ProgramPath +
    @"\\DeleteData.TDD");
```

Consider what would happen if all the files in the delete list were past their expiry date and had all been deleted. In this case there would be no .T\$\$ file to rename to .TDD, and in fact the .TDD file should no longer exist. The solution here is another try-catch:

```
try
{
    System.IO.File.Move(ProgramPath +
        @"\\DeleteData.T$$",
        ProgramPath +
        @"\\DeleteData.TDD");
}
catch { }
```

There are probably other similar, rarely occurring conditions that need to be handled, but this is what extended debugging in a beta test phase is all about. With the addition of these try-catch clauses, the program doesn't crash. **CS**